COS 318: Operating Systems

Processes and Threads

Prof. Margaret Martonosi
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall11/cos318

Today's Topics

- Processes
- Concurrency
- Threads

- Reminder:
  - Hope you're all busy implementing your assignment

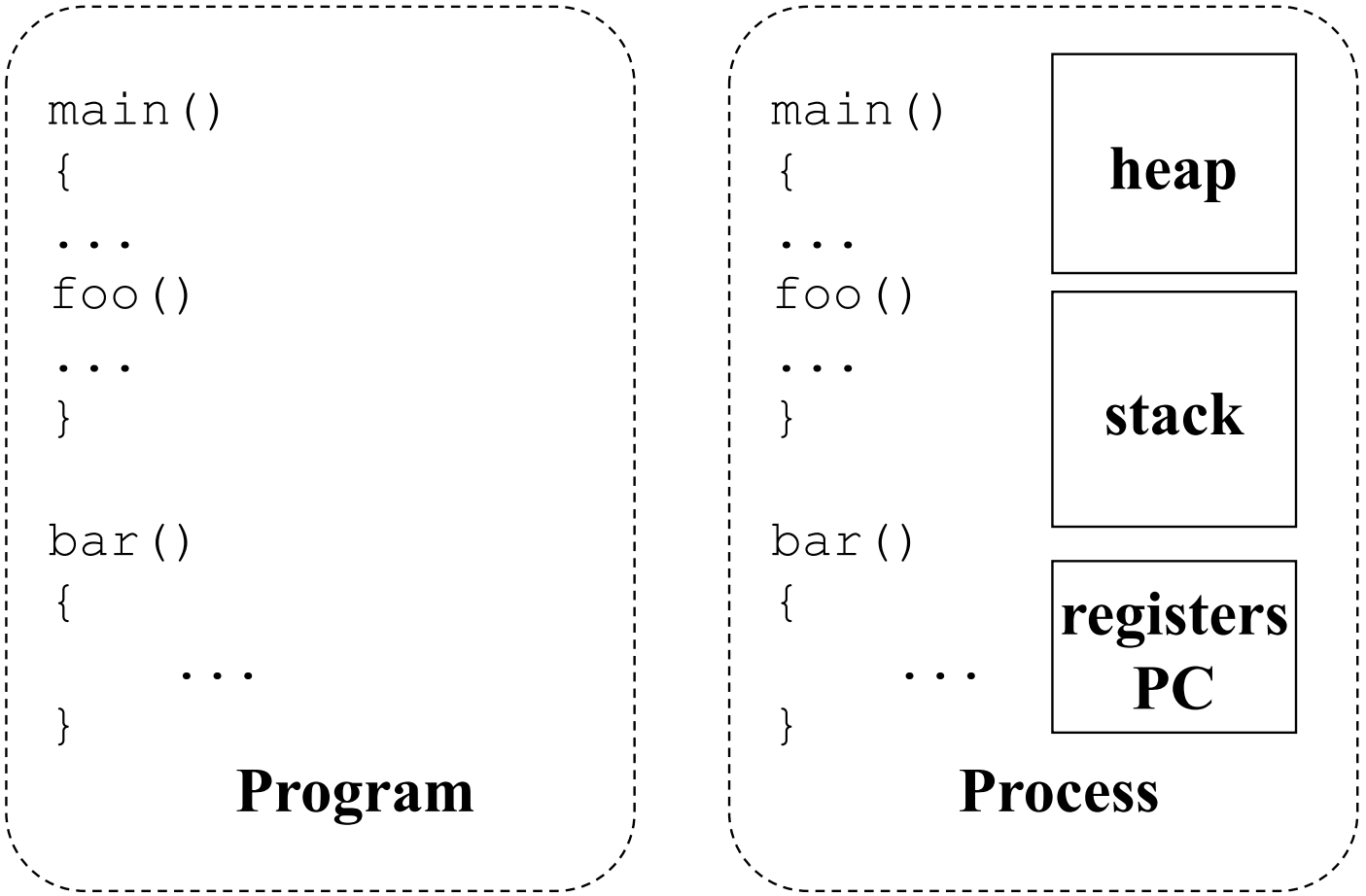
(Traditional) OS Abstractions

- Processes - thread of control with context
- Files- In Unix, this is “everything else”
  - Regular file – named, linear stream of data bytes
  - Sockets - endpoints of communication, possible between unrelated processes
  - Pipes - unidirectional I/O stream, can be unnamed
  - Devices

Process

- Most fundamental concept in OS
- Process: a program in execution
  - one or more threads (units of work)
  - associated system resources
- Program vs. process
  - program: a passive entity
  - process: an active entity
- For a program to execute, a process is created for that program

Program and Process



Process vs. Program

- Process > program
  - Program is just part of process state
  - Example: many users can run the same program
    - Each process has its own address space, i.e., even though program has single set of variable names, each process will have different values
- Process < program
  - A program can invoke more than one process
  - Example: Fork off processes

Simplest Process

- Sequential execution
  - No concurrency inside a process
  - Everything happens sequentially
  - Some coordination may be required
- Process state
  - Registers
  - Main memory
  - I/O devices
    - File system
    - Communication ports
  - …

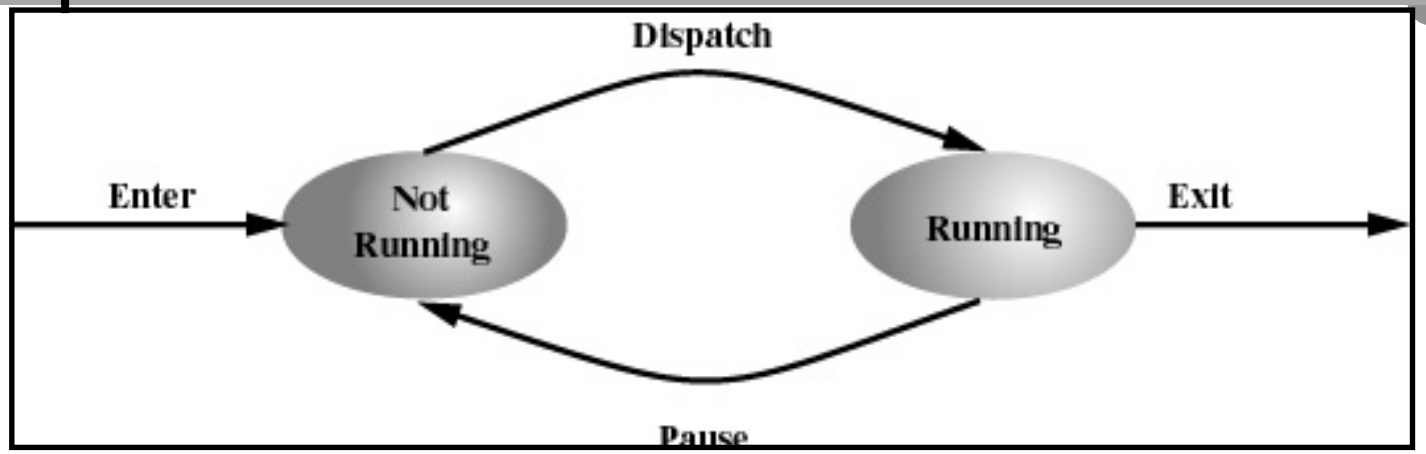
Process Abstraction

- Unit of scheduling
- One (or more*) sequential threads of control
  - program counter, register values, call stack
- Unit of resource allocation
  - address space (code and data), open files
  - sometimes called tasks or jobs
- Operations on processes: fork (clone-style creation), wait (parent on child),
  exit (self-termination), signal, kill.

Process Management

- Fundamental task of any OS
- For processes, OS must:
  - allocate resources
  - facilitate multiprogramming
  - allow sharing and exchange of info
  - protect resources from other processes
  - enable synchronization
- How?
  - data structure for each process
  - describes state and resource ownership

Process Scheduling: A Simple Two-State Model

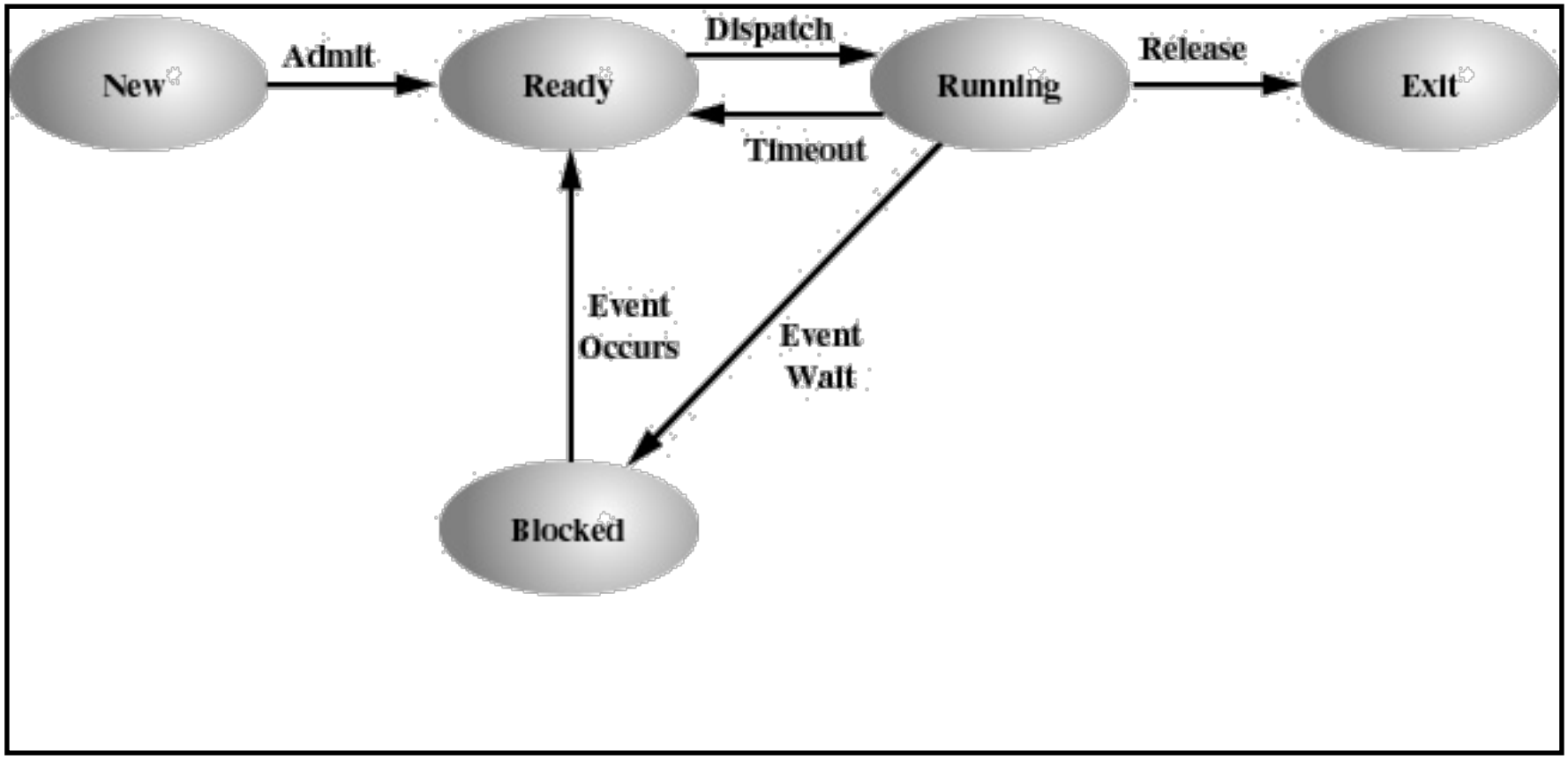


- What are the two simplest states of a process?
  - Running
  - Not running
- When a new process created: “not running”
  - memory allocated, enters waiting queue
- Eventually, a “running” process is interrupted
  - state is set to “not running”
- Dispatcher chooses another from queue
  - state is set to “running” and it executes

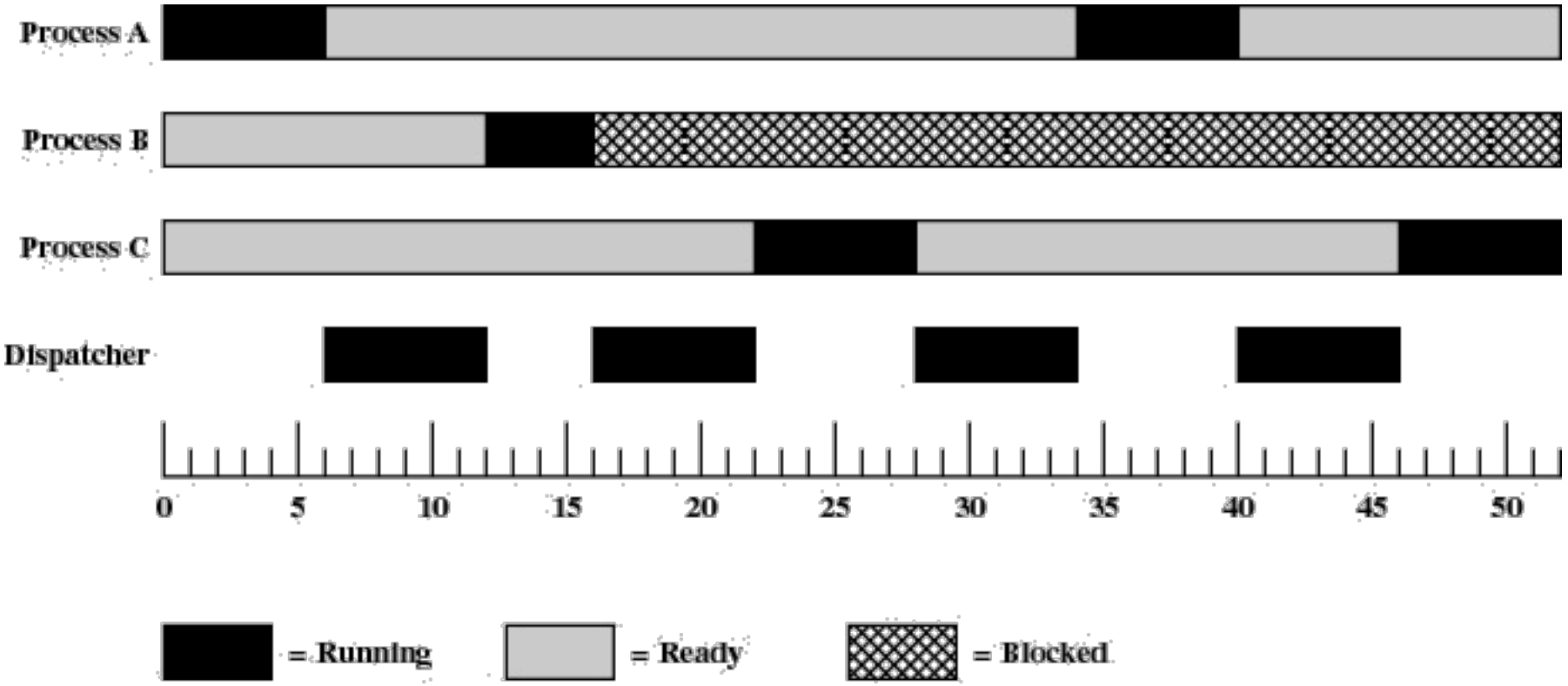
Two States: Not Enough

- Running process makes I/O syscall
  - moved to “not running” state
  - can’t be selected until I/O is complete!
- “not running” should be two states:
  - blocked: waiting for something, can’t be selected
  - ready: just itching for CPU time…
- Five states total
  - running, blocked, ready
  - new: OS might not yet admit (e.g., performance)
  - exiting: halted or aborted
    - perhaps other programs want to examine tables & DS

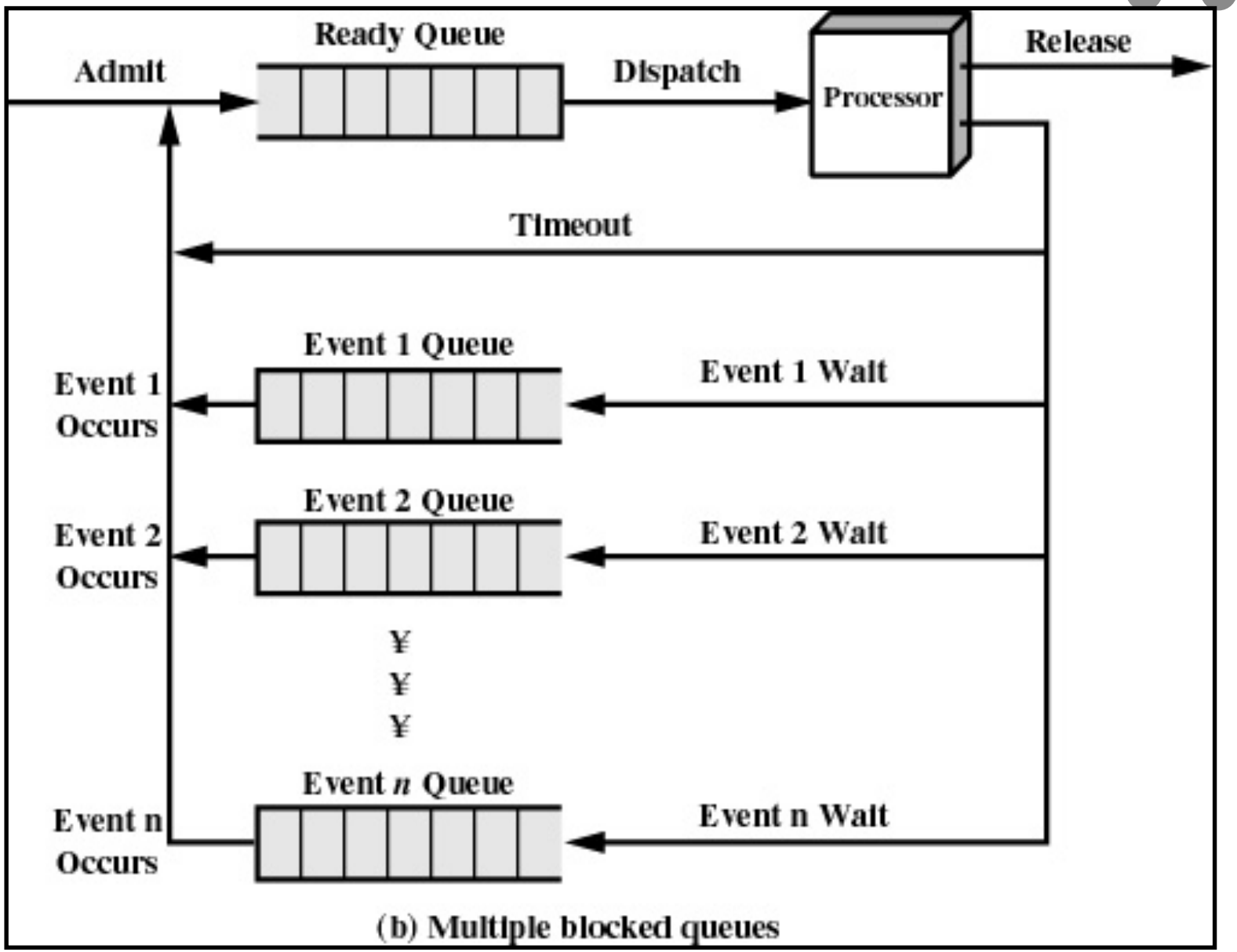
The Five-State Model



Process State Diagram



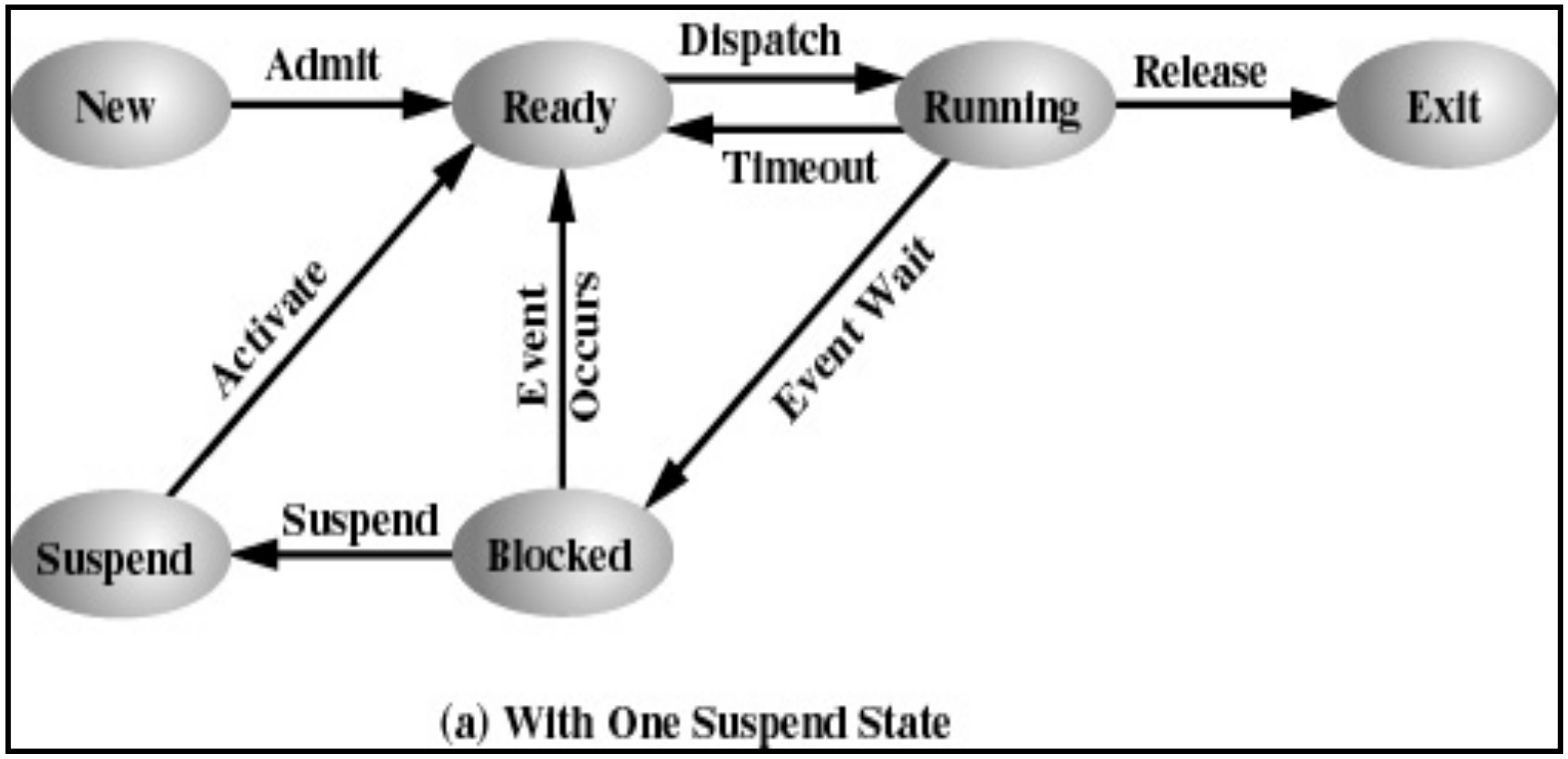
OS Queuing Diagram



Are Five States Enough?

- Problem: Can't have all processes in RAM
- Solution: swap some to disk
  - i.e., move all or part of a process to disk
- Requires new state: suspend
  - on disk, therefore not available to CPU

Six-State Model



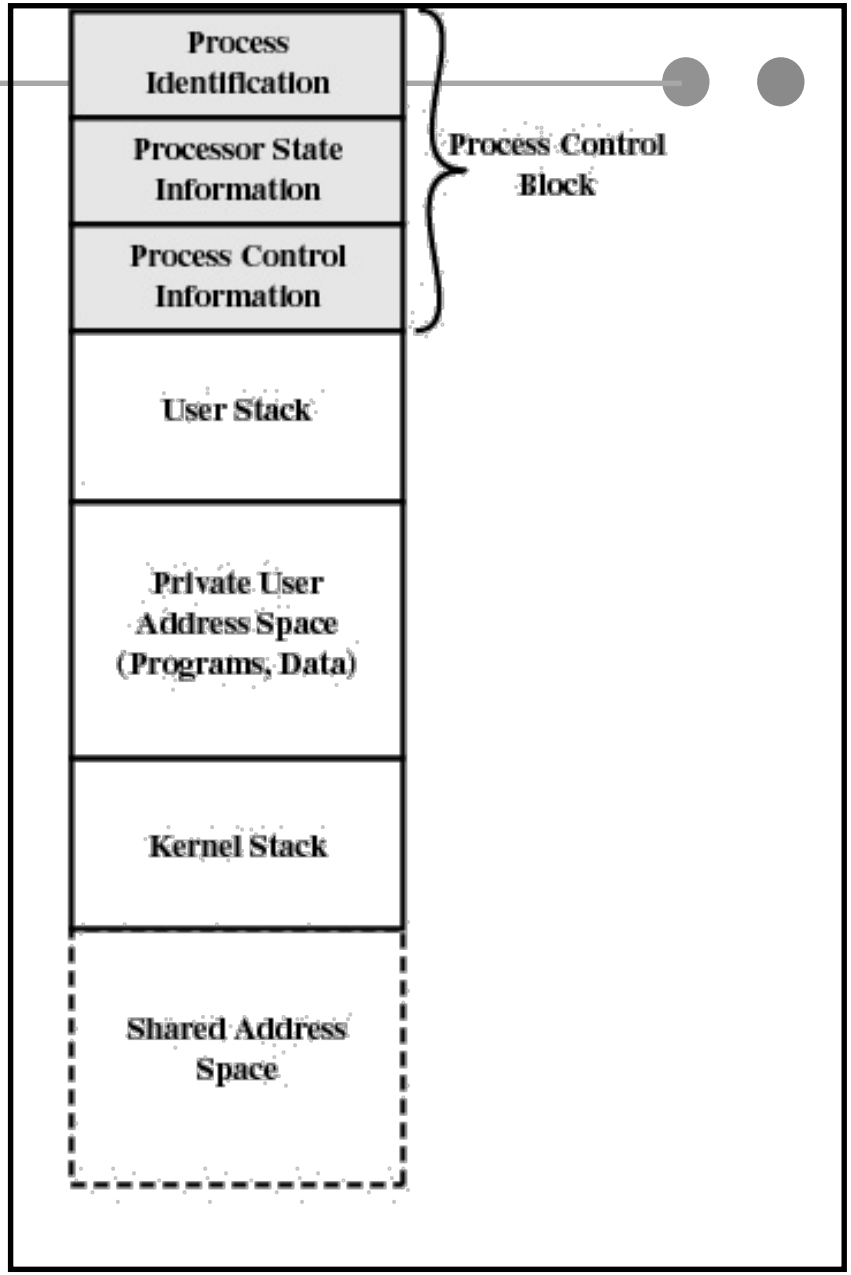
(a) With One Suspend State

Process Image

- Must know:
  - where process is located
  - attributes for managing
- Process image: physical manifestation of process
  - program(s) to be executed
  - data locations for vars and constants
  - stack for procedure calls and parameter passing
  - PCB: info used by OS to manage

Process Image

- At least small portion must stay in RAM



Process Control Block (PCB)

- Process management info
  - State
    - Ready: ready to run
    - Running: currently running
    - Blocked: waiting for resources
  - Registers, EFLAGS, and other CPU state
  - Stack, code and data segment
  - Parents, etc
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.
- How OS takes care of processes
  - Resource allocation and process state transition

Primitives of Processes

- Creation and termination
  - Exec, Fork, Wait, Kill
- Signals
  - Action, Return, Handler
- Operations
  - Block, Yield
- Synchronization
  - We will talk about this later

Make A Process

- Creation
  - Load code and data into memory
  - Create an empty call stack
  - Initialize state to same as after a process switch
  - Make the process ready to run
- Clone
  - Stop current process and save state
  - Make copy of current code, data, stack and OS state
  - Make the process ready to run

Process Creation

- Assign a new process ID
  - new entry in process table
- Allocate space for process image
  - space for PCB
  - space for address space and user stack
- Initialize PCB
  - ID of process, parent
  - PC set to program entry point
  - typically, “ready” state
- Linkages and other DS
  - place image in list/queue
  - accounting DS

Or clone from another process

Example: Unix

- How to make processes:
  - fork clones a process
  - exec overlays the current process

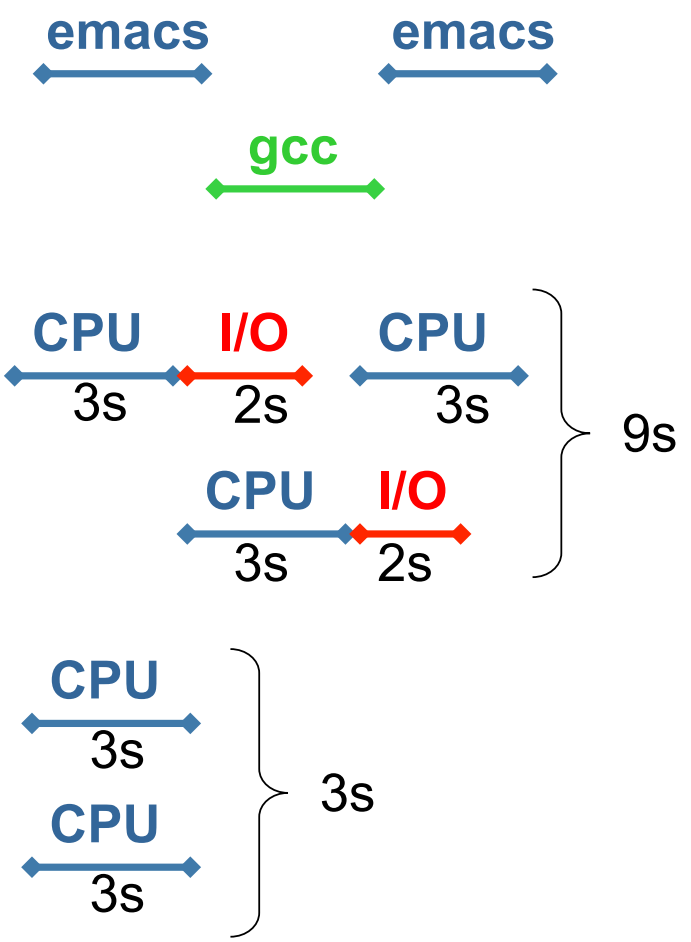
```
If ((pid = fork()) == 0) {
      /* child process */
     exec("foo");  /* does not return */
else
     /* parent */
     wait(pid);    /* wait for child to die */
```

Concurrency and Process

- Concurrency
  - Hundreds of jobs going on in a system
  - CPU is shared, as are I/O devices
  - Each job would like to have its own computer
- Process concurrency
  - Decompose complex problems into simple ones
  - Make each simple one a process
  - Deal with one at a time
  - Each process feels like having its own computer
- Example: gcc (via “gcc –pipe –v”) launches
  - /usr/libexec/cpp | /usr/libexec/cc1 | /usr/libexec/as | /usr/libexec/elf/ld
  - Each instance is a process

Process Parallelism

- Virtualization
  - Each process run for a while
  - Make a CPU into many
  - Each virtually has its own CPU
- I/O parallelism
  - CPU job overlaps with I/O
  - Each runs almost as fast as if it has its own computer
  - Reduce total completion time
- CPU parallelism
  - Multiple CPUs (such as SMP)
  - Processes running in parallel
  - Speedup



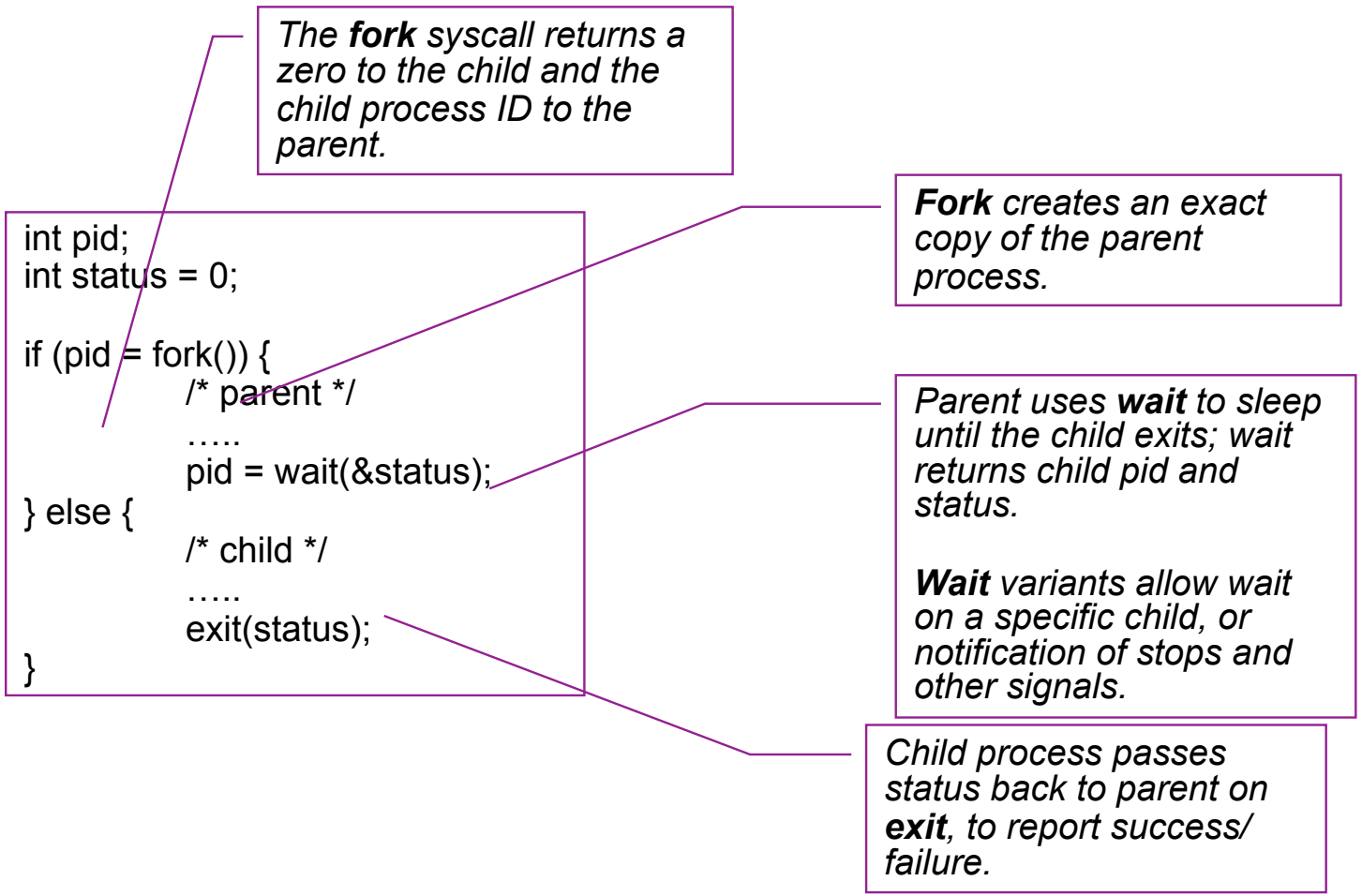
More on Process Parallelism

- Process parallelism is common in real life
  - Each sales person sell $1M annually
  - Hire 100 sales people to generate $100M revenue
- Speedup
  - Ideal speedup is factor of N
  - Reality: bottlenecks + coordination overhead
- Question
  - Can you speedup by working with a partner?
  - Can you speedup by working with 20 partners?
  - Can you get super-linear (more than a factor of N) speedup?

Process-related System Calls

- Simple and powerful primitives for process creation and initialization.
  - Unix ***fork*** creates a *child* process as (initially) a clone of the parent [Linux: fork() implemented by clone() system call]
  - parent program runs in child process – maybe just to set it up for ***exec***
  - child can ***exit***, parent can ***wait*** for child to do so. [Linux: wait4 system call]
- Rich facilities for controlling processes by asynchronous *signals*.
  - notification of internal and/or external events to processes or groups
  - the look, feel, and power of interrupts and exceptions
  - default actions: stop process, kill process, dump core, no effect
  - user-level handlers

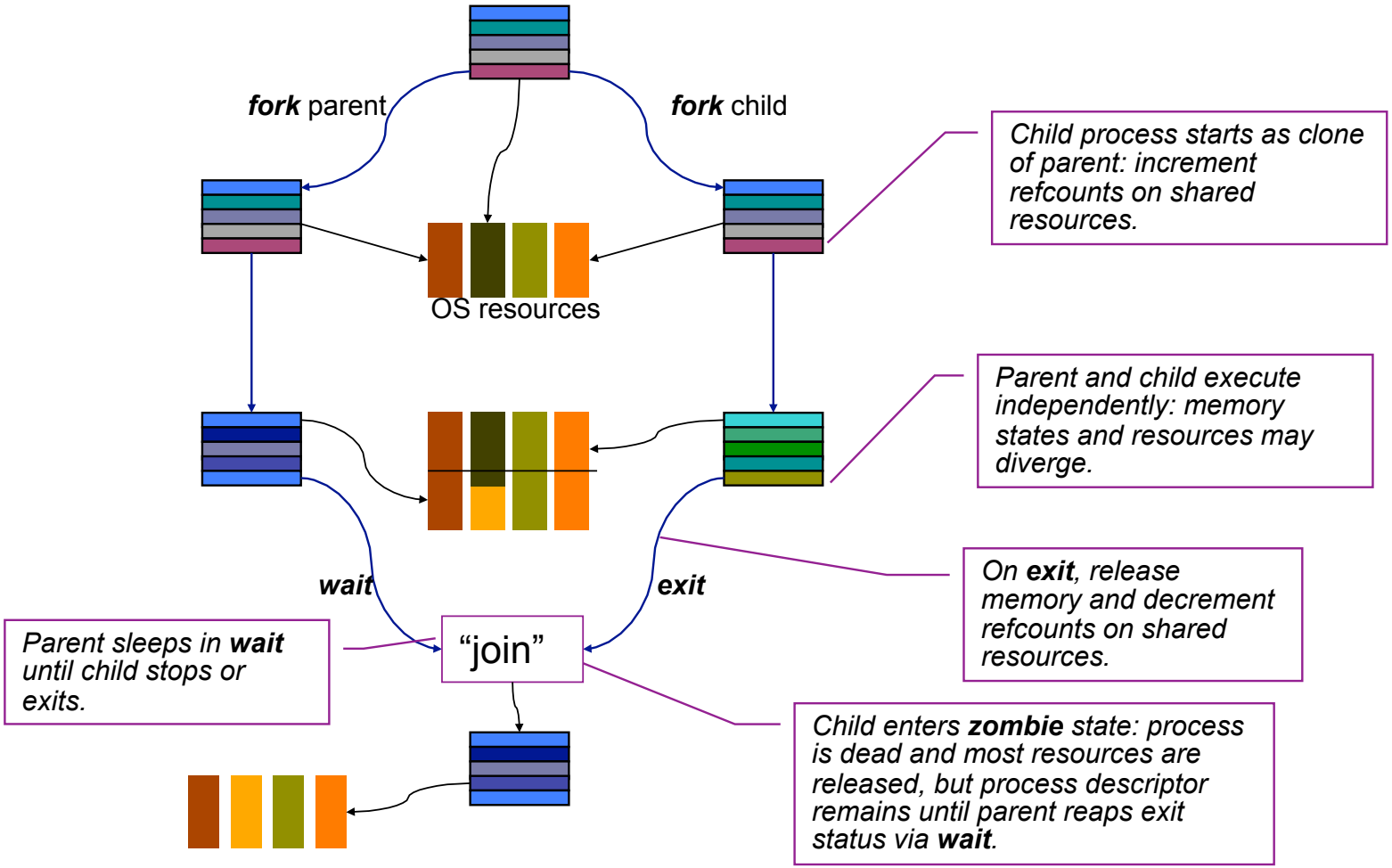
Process Control



Child Discipline

- After a fork, the parent program (not process) has complete control over the behavior of its child process.
- The child inherits its execution environment from the parent...but the parent program can change it.
  - sets bindings of file descriptors with open, close, dup
  - pipe sets up data channels between processes
- Parent program may cause the child to execute a different program, by calling exec* in the child context.

Fork/Exit/Wait Example

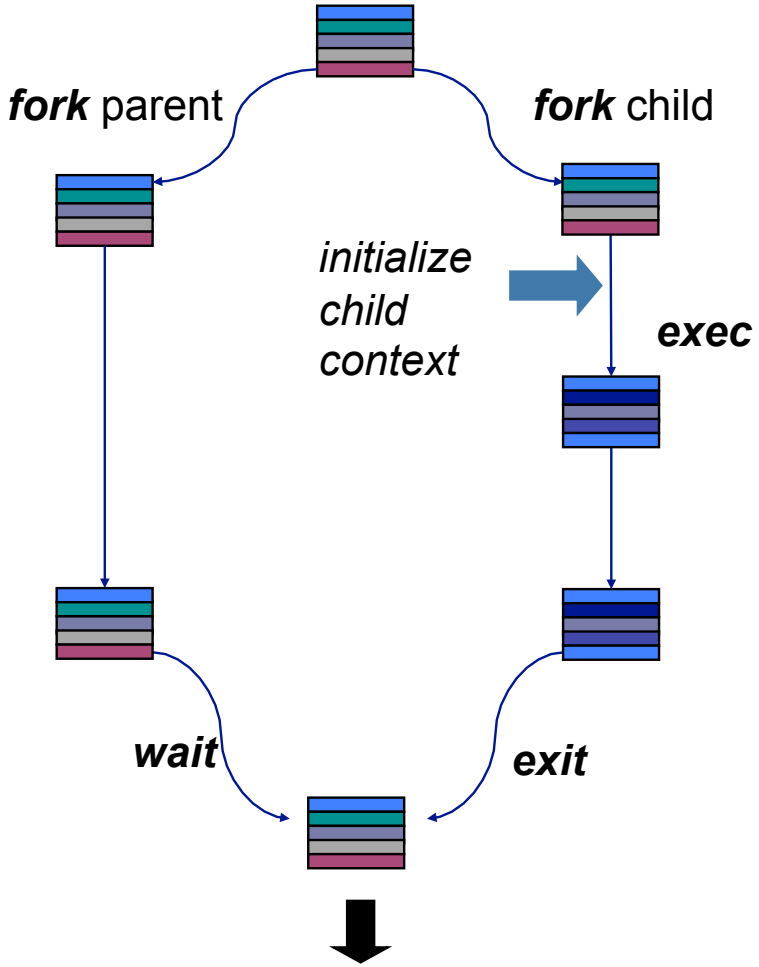


Why are reference counts needed on shared resources?

Exec, Execve, etc.

- Children should have lives of their own.
- Exec* “boots” the child with a different executable image.
  - parent program makes exec* syscall (in forked child context) to run a program in a new child process
  - exec* overlays child process with a new executable image
  - restarts in user mode at predetermined entry point (e.g., crt0)
  - no return to parent program (it’s gone)
  - arguments and environment variables passed in memory
  - file descriptors etc. are unchanged

Fork/Exec/Exit/Wait Example



int pid = fork();

Create a new process that is a clone of its parent.

exec*("program" *[, argvp, envp]*);

Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);

Exit with status, destroying the process.

int pid = wait*(&status);

Wait for exit (or other status change) of a child.

Join Scenarios

- Several cases must be considered for join (e.g., exit/wait).
  - What if the child exits before the parent does the wait?
    - "Zombie" process object holds child status and stats.
  - What if the parent continues to run but never joins?
    - Danger of filling up memory with zombie processes?
    - Parent might have specified it was not going to wait or that it would ignore its child's exit. Child status can be discarded.
  - What if the parent exits before the child?
    - Orphans become children of init (process 1).
  - What if the parent can't afford to get "stuck" on a join?
    - Asynchronous notification (we'll see an example later).

Linux Processes

- Processes and threads are not differentiated – with varying degrees of shared resources
- clone() system call takes flags to determine what resources parent and child processes will share:
  - Open files
  - Signal handlers
  - Address space
  - Same parent

Process Context Switch

- Save a context (everything that a process may damage)
  - All registers (general purpose and floating point)
  - All co-processor state
  - Save all memory to disk?
  - What about cache and TLB stuff?
- Start a context
  - Does the reverse
- Challenge
  - OS code must save state without changing any state
  - How to run without touching any registers?
    - CISC machines have a special instruction to save and restore all registers on stack
    - RISC: reserve registers for kernel or have way to carefully save one and then continue

Today's Topics

- Processes
- Concurrency
- Threads

Before Threads…

- Recall that a process consists of:
  - program(s)
  - data
  - stack
  - PCB
- all stored in the process image

- Process (context) switch is pure overhead

Process Characterization

- Process has two characteristics:
  - resource ownership
    - address space to hold process image
    - I/O devices, files, etc.
  - execution
    - a single execution path (thread of control)
    - execution state, PC & registers, stack

Refining Terminology

- Distinguish the two characteristics
  - process: resource ownership
  - thread: unit of execution (dispatching)
    - AKA lightweight process (LWP)
- Multi-threading: support multiple threads of execution within a single process
- Process, as we have known it thus far, is a single-threaded process

Threads

- Thread
  - A sequential execution stream within a process (also called lightweight process)
  - Threads in a process share the same address space
- Thread concurrency
  - Easier to program I/O overlapping with threads than signals
  - Responsive user interface
  - Run some program activities “in the background”
  - Multiple CPUs sharing the same memory

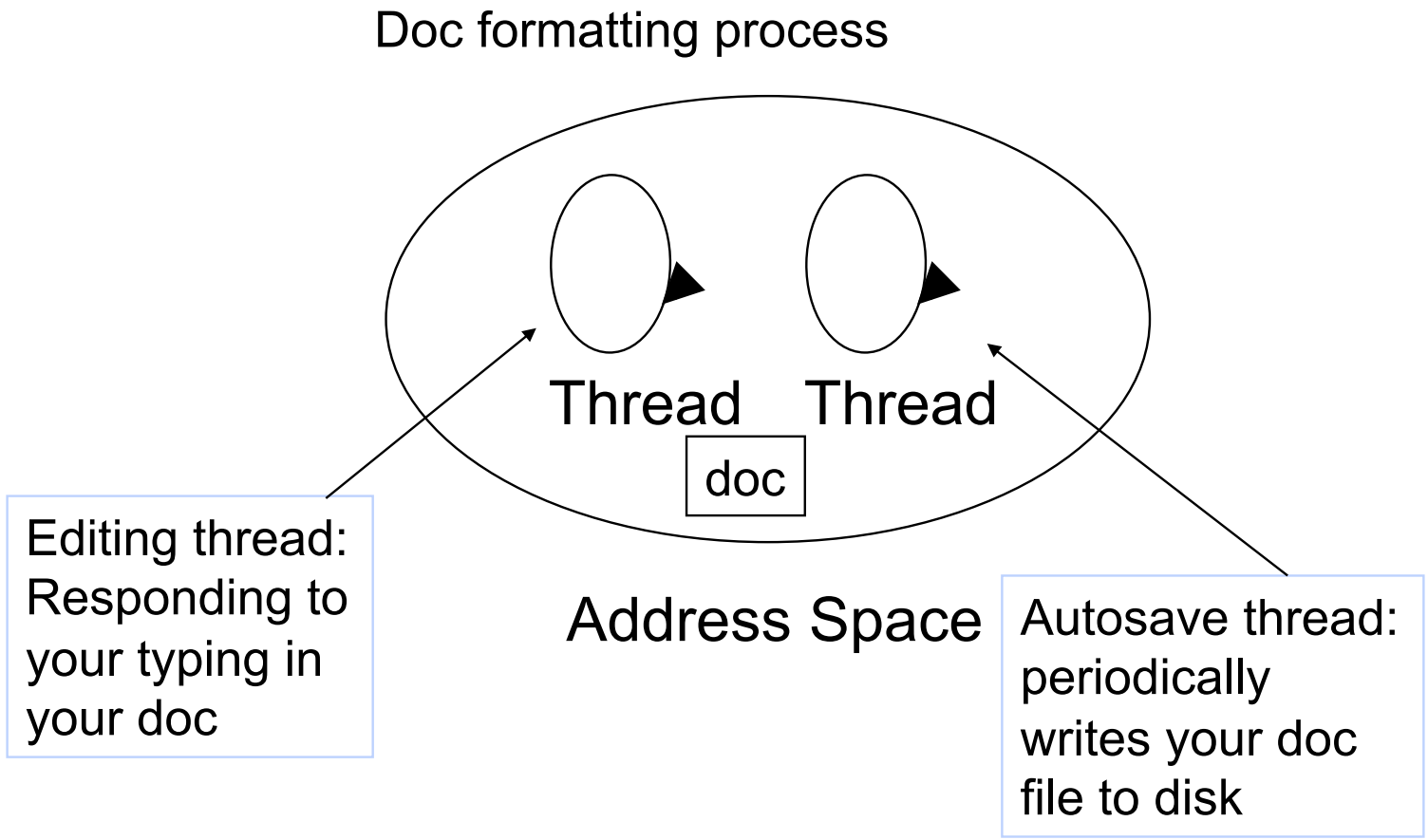
Threads and Processes

- Decouple the resource allocation aspect from the control aspect
- Thread abstraction - defines a single sequential instruction stream (PC, stack, register values)
- Process - the resource context serving as a “container” for one or more threads (shared address space)

Process vs. Threads

- Address space
  - Processes do not usually share memory
  - Process context switch changes page table and other memory mechanisms
  - Threads in a process share the entire address space
- Privileges
  - Processes have their own privileges (file accesses, e.g.)
  - Threads in a process share all privileges
- Question
  - Do you really want to share the “entire” address space?

An Example


Doc formatting process
Thread
Thread
doc
Address Space
Editing thread: Responding to your typing in your doc
Autosave thread: periodically writes your doc file to disk

Thread Control Block (TCB)

- State
  - Ready: ready to run
  - Running: currently running
  - Blocked: waiting for resources
- Registers
- Status (EFLAGS)
- Program counter (EIP)
- Stack
- Code

Typical Thread API

- Creation
  - Create, Join, Exit
- Mutual exclusion
  - Acquire (lock), Release (unlock)
- Condition variables
  - Wait, Signal, Broadcast

Thread Context Switch

- Save a context (everything that a thread may damage)
  - All registers (general purpose and floating point)
  - All co-processor state
  - Need to save stack?
  - What about cache and TLB stuff?
- Start a context
  - Does the reverse
- May trigger a process context switch

Procedure Call

- Caller or callee save some context (same stack)
- Caller saved example:

Threads vs. Procedures

- Threads may resume out of order
  - Cannot use LIFO stack to save state
  - Each thread has its own stack
- Threads switch less often
  - Do not partition registers
  - Each thread “has” its own CPU
- Threads can be asynchronous
  - Procedure call can use compiler to save state synchronously
  - Threads can run asynchronously
- Multiple threads
  - Multiple threads can run on multiple CPUs in parallel
  - Procedure calls are sequential

Multi-Threaded Environment

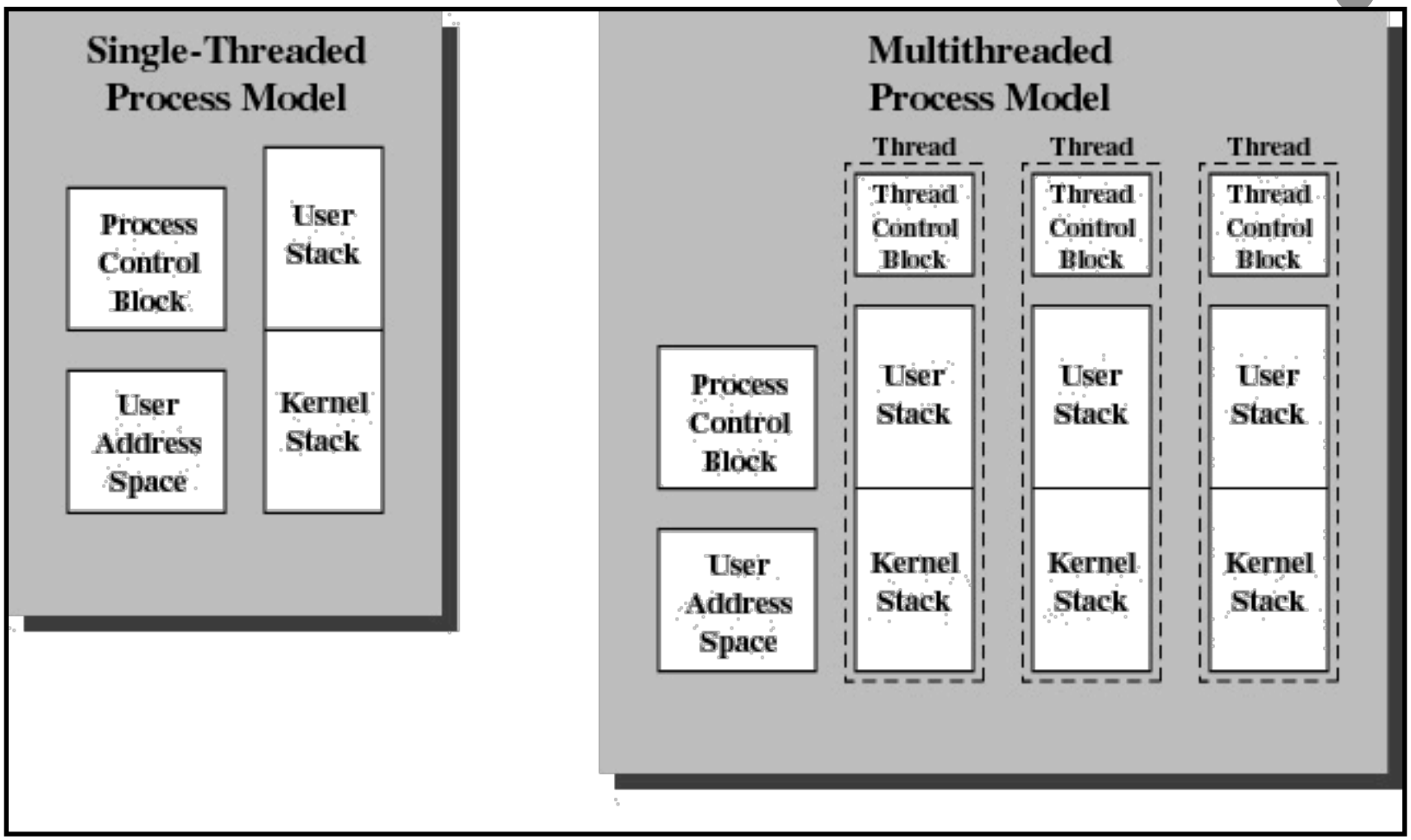
- Process:
  - virtual address space (for image)
  - protected access to resources
    - processors, other processes, I/O, files
- Thread: one or more w/in a process
  - execution state
  - saved context when not running (i.e., independent PC)
  - stack
  - access to memory & resources of the process

Multi-Threaded Environment

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

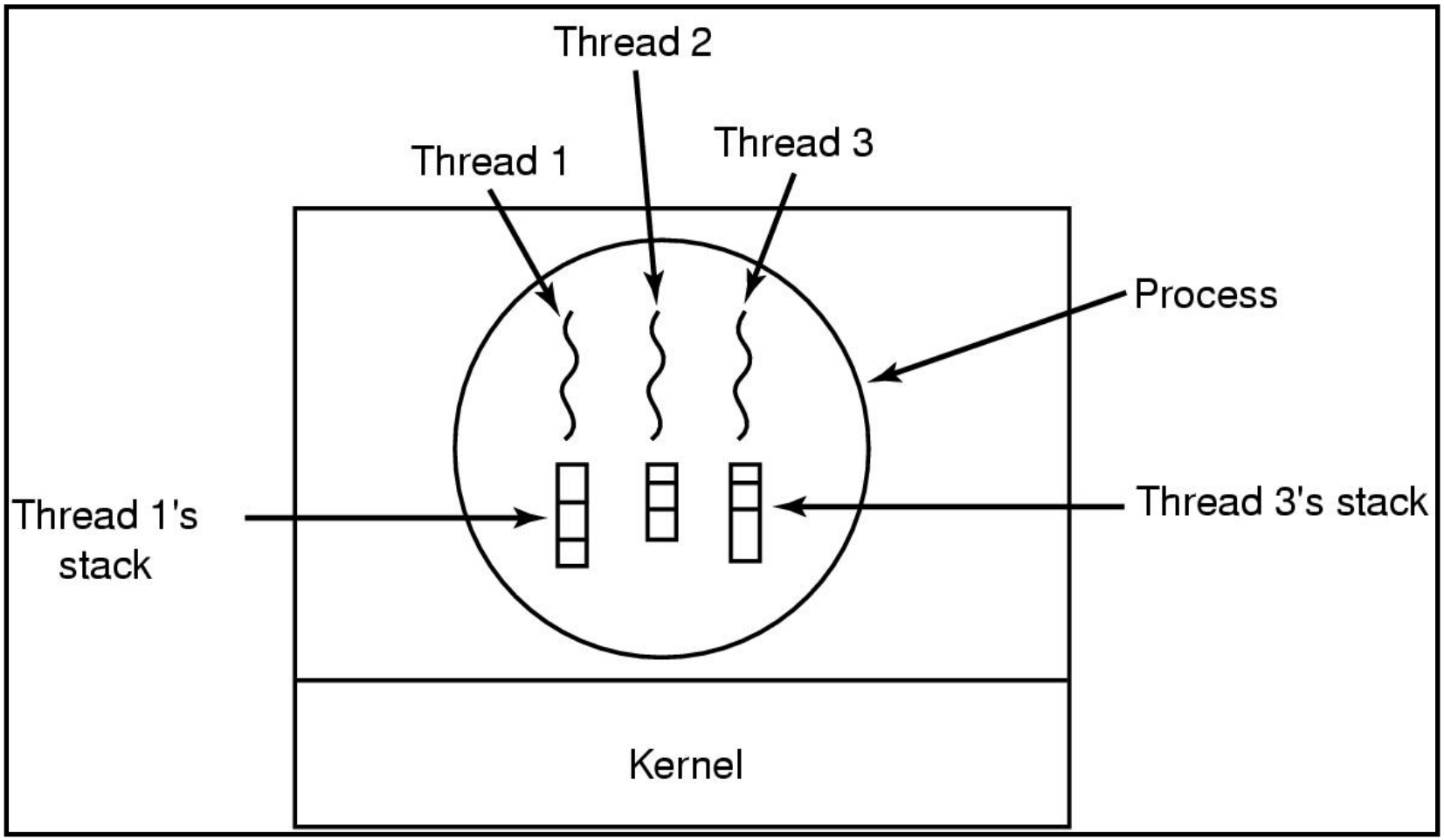
- Left: shared by all threads in a process
- Right: private to each thread

Single- vs. Multi-threaded Model



- still a single PCB & addr space per process
- separate stacks, TCB for each thread

Single- vs. Multi-threaded Model



Remember…

- Different threads in a process have same address space
- Every thread can access every mem addr w/in addr space
  - No protection between threads
- Each thread has its own stack
  - one frame per procedure called but not completed (local vars, return address)

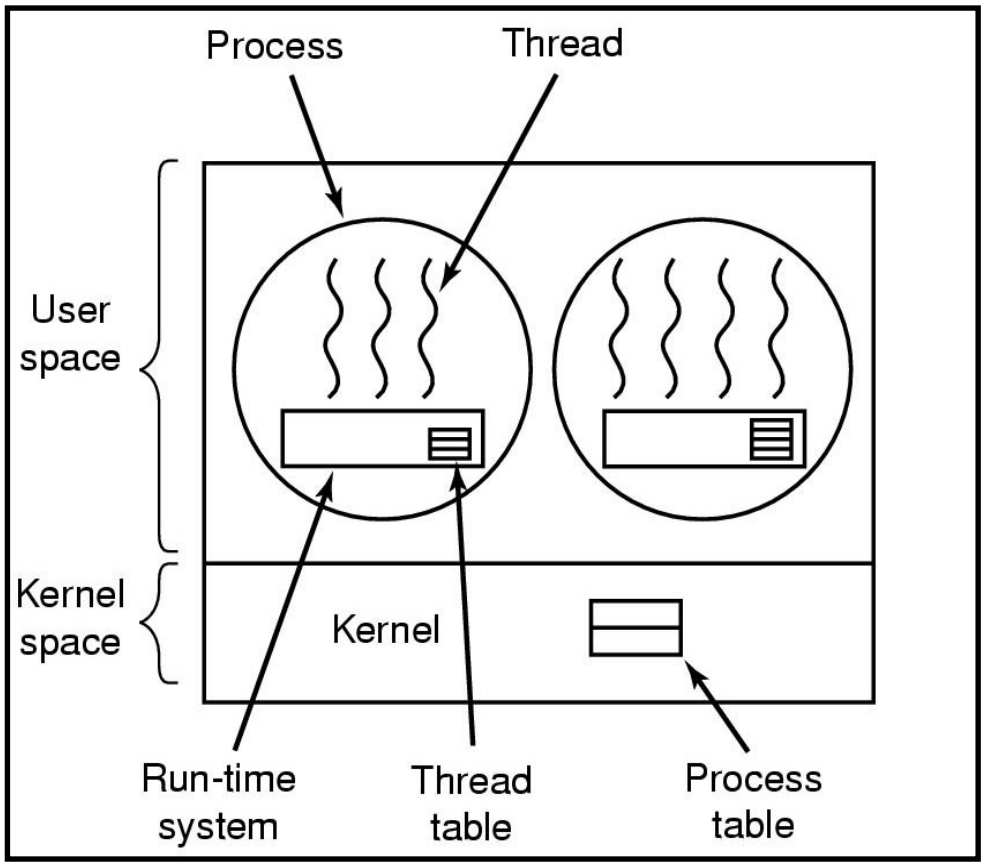
Why Threads?

- In many apps, multiple activities @ once
  - e.g., word processor
- Easier to create and destroy than processes
  - no resources attached to threads
- Allow program to continue if part is blocked
  - permit I/O- and CPU-bound activities to overlap
  - speeds up application
- Easy resource sharing (same addr space!)
- Take advantage of multiprocessors

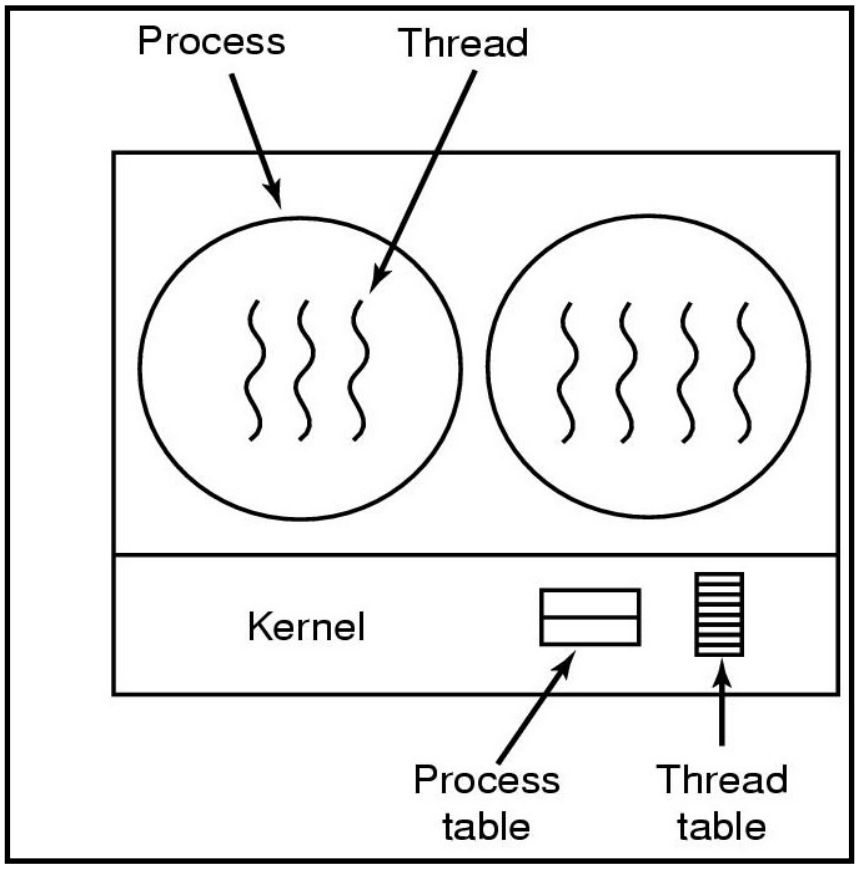
Thread Functionality

- Scheduling done on a per-thread basis
- Terminate process --> kill all threads
- Four basic thread operations:
  - spawn (automatically spawned for new process)
  - block
  - unblock
  - terminate
- Synchronization:
  - all threads share same addr space & resources
  - must synchronize to avoid conflicts
  - process synchro techniques are same for threads (later)

Two Types Of Threads



User-Level



Kernel-Level

User-Level Threads

- Thread management done by an application
- Use thread library (e.g., POSIX Pthreads)
  - create/destroy, pass msgs, schedule execution, save/restore contexts
- Each process needs its own thread table
- Kernel is unaware of these threads
  - assigns single execution state to the process
  - unaware of any thread scheduling activity

User-Level Threads

- Advantages:
  - thread switch does not require kernel privileges
  - thread switch more efficient than kernel call
  - scheduling can be process (app) specific
    - without disturbing OS
  - can run on any OS
  - scales easily

- Disadvantages:
  - if one thread blocks, all are blocked (process switch)
    - e.g., I/O, page faults
  - cannot take advantage of multiprocessor
    - one process to one processor
  - programmers usually want threads for blocking apps

Kernel-Level Threads

- Thread management done by kernel
  - process as a whole (process table)
  - individual threads (thread table)
- Kernel schedules on a per-thread basis
- Addresses disadvantages of ULT:
  - schedule multi threads from one process on multiple CPUs
  - if one thread blocks, schedule another (no process switch)
- Disadvantage of KLT:
  - thread switch causes mode switch to kernel

Real Operating Systems

- One or many address spaces
- One or many threads per address space

	1 address space	Many address spaces
1 thread per address space	MSDOS Macintosh	Traditional Unix
Many threads per address spaces	Embedded OS, Pilot	VMS, Mach (OS-X), OS/2, Windows NT/XP/Vista, Solaris, HP-UX, Linux

Summary

- Concurrency
  - CPU and I/O
  - Among applications
  - Within an application
- Processes
  - Abstraction for application concurrency
- Threads
  - Abstraction for concurrency within an application

Unix Signals

- Signals notify processes of internal or external events.
  - the Unix software equivalent of interrupts/ exceptions
  - only way to do something to a process “from the outside”
  - Unix systems define a small set of signal types
- Examples of signal generation:
  - keyboard ***ctrl-c*** and ***ctrl-z*** signal the *foreground process*
  - synchronous fault notifications, sysc
  - asynchronous notifications from other processes via ***kill***
  - IPC events (SIGPIPE, SIGCHLD)

signal == “upcall”

Process Handling of Signals

1. Each signal type has a system-defined default action.
    - abort and dump core (SIGSEGV, SIGBUS, etc.)
    - ignore, stop, exit, continue
2. A process may choose to *block* (inhibit) or *ignore* some signal types.
3. The process may choose to *catch* some signal types by specifying a (user mode) *handler* procedure.
    - specify alternate signal stack for handler to run on
    - system passes interrupted context to handler
    - handler may munge and/or return to interrupted context

Predefined Signals (a Sampler)

Name	Default action	Description
SIGINT	Quit	Interrupt
SIGILL	Dump	Illegal instruction
SIGKILL	Quit	Kill (can not be caught, blocked, or ignored
SIGSEGV	Dump	Out of range addr
SIGALRM	Quit	Alarm clock
SIGCHLD	Ignore	Child status change
SIGTERM	Quit	Sw termination sent by kill

User's View of Signals

```
int alarmflag=0;
alarmHandler ()
{    printf("An alarm clock signal was received\n");
     alarmflag = 1;
 }
main()
{
     signal (SIGALRM, alarmHandler);
     alarm(3); printf("Alarm has been set\n");
     while (!alarmflag) pause ();
     printf("Back from alarm signal handler\n");
}
```

Sets up signal handler

Instructs kernel to send SIGALRM in 3 seconds

Suspends caller until signal

User's View of Signals II

```
main()
{
    int (*oldHandler) ();
    printf ("I can be control-c'ed\n");
    sleep (3);
    oldHandler = signal (SIGINT, SIG_IGN);
    printf("I'm protected from control-c\n");
    sleep(3);
    signal (SIGINT, oldHandler);
    printf("Back to normal\n");
    sleep(3); printf("bye\n");
}
```

Yet Another User's View

```
main(argc, argv)
int argc; char* argv[];
{
        int pid;
        signal (SIGCHLD,childhandler);
        pid = fork ();
        if (pid == 0) /*child*/
        { execvp (argv[2], &argv[2]); }
        else
        {sleep (5);
         printf("child too slow\n");
         kill (pid, SIGINT);
        }
}
```

```
childhandler()
{       int childPid, childStatus;
        childPid = wait (&childStatus);
        printf("child done in time\n");
        exit;
 }
```

Collects status

SIGCHLD sent
by child on termination;
if SIG_IGN, dezombie

What does this do?

The Basics of Processes

- Processes are the OS-provided abstraction of multiple tasks (including user programs) executing concurrently.
- Program = a passive set of bits
- Process = 1 instance of that program as it executes
  - => has an execution context – register state, memory resources, etc.)
- OS schedules processes to share CPU.

Process State Transition